SCR v1.1.8 User Manual

September 28, 2015

Abstract

The Scalable Checkpoint / Restart (SCR) library enables MPI applications to utilize distributed storage on Linux clusters to attain high file I/O bandwidth for checkpointing and restarting large-scale jobs. With SCR, jobs run more efficiently, recompute less work upon a failure, and reduce load on critical shared resources such as the parallel file system.

SCR caches checkpoint files in storage local to the compute nodes, and it applies a redundancy scheme such that files can be recovered in the event of a failure. When a failure occurs, the current run is killed. If there are sufficient spare nodes and time remaining in the resource allocation, another run may be restarted from the cached checkpoint files. Otherwise, the cached checkpoint files are flushed to the parallel file system, and the resource allocation is released. With this approach, file I/O bandwidth scales linearly with the number of compute nodes. On large clusters, SCR is often 100x to 1000x faster than the parallel file system.

The SCR library is written in C, and it provides C and Fortran interfaces. It has been used in production at LLNL since 2007 using RAM disk and SSDs on Linux / x86-64 / Infiniband clusters. The original implementation required the SLURM resource manager, but it was designed to be portable. In 2011, SCR was ported to Cray XT platforms running the TORQUE resource manager. SCR is an open source project under a BSD license hosted at: `https://github.com/hpc/scr`.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344.

Contents

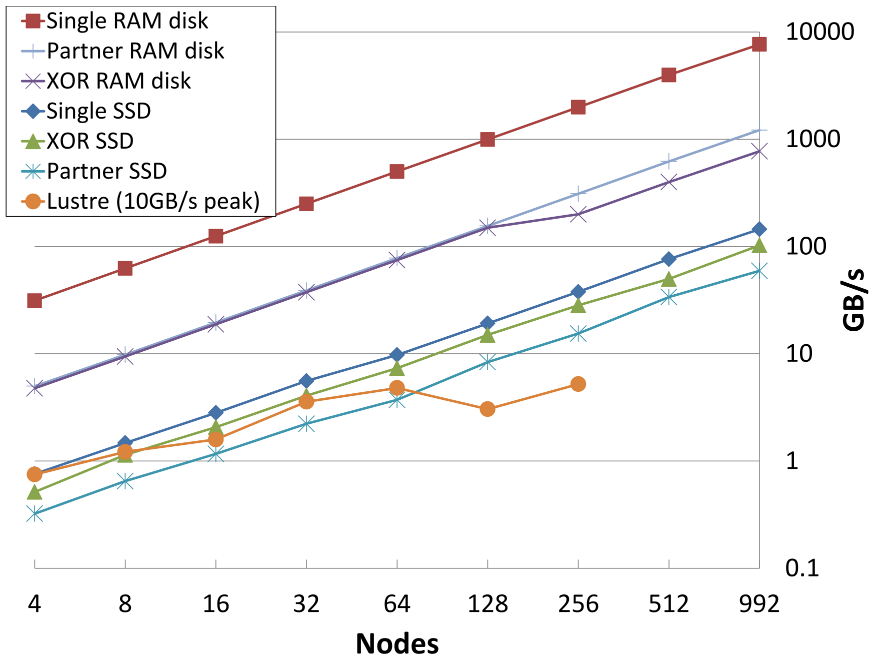

Figure 1: Aggregate checkpoint write bandwidth on Coastal

1 Introduction

Compared to small-scale jobs, large-scale jobs face an increased likelihood of encountering system failures during their run. To handle more frequent failures, large-scale jobs must checkpoint more frequently. These checkpoints ultimately must be stored in the parallel file system, but checkpointing a large-scale job to the parallel file system has two drawbacks. First, large-scale jobs must often checkpoint large data sets for which the parallel file system delivers a relatively small amount of bandwidth. Second, the parallel file system is often shared among multiple jobs, and so it may be busy servicing other jobs when the job needs to checkpoint. Either condition idles the job while it waits for the storage resource.

The Scalable Checkpoint / Restart (SCR) library addresses both problems by caching checkpoints in scalable, but less reliable storage. The solution derives from two key observations. First, a job only needs its most recent checkpoint. As soon as the next checkpoint is written, the previous checkpoint can be discarded. Second, a typical failure disables a small portion of the system, but it otherwise leaves most of the system intact. Leveraging these two properties, SCR caches only the most recent checkpoints, and it applies a redundancy scheme such that checkpoints can be recovered after a failure disables a small portion of the system. Each checkpoint is written to the cache and discarded, unless a failure occurs, at which point SCR recovers the most recent checkpoint from cache. It then copies the recovered checkpoint to the parallel file system, or if there are spare resources available, the job restarts directly from its cached checkpoint. With this design, SCR delivers scalable bandwidth utilizing storage resources that are fully dedicated to the job. The current implementation significantly outperforms the parallel file system (see Figure 1).

Even when using SCR, some checkpoints must be written to the parallel file system, because some failures disable larger portions of the system than the redundancy scheme can handle. However, with a well-chosen redundancy scheme, the frequency with which these writes must be made can be greatly reduced. In essence, SCR is a production-level implementation of a two-level checkpoint system of the type analyzed by Vaidya in [1].

Because large-scale jobs checkpoint faster with SCR, it is possible to configure a job to checkpoint more frequently while simultaneously reducing its checkpointing overhead. Such a job then loses less work upon a failure, but it also utilizes the machine more efficiently when there are no failures. Additionally, if spare resources are available, a job can be restarted faster with SCR. Finally, by shifting its checkpointing traffic to SCR, a job reduces its load on the parallel file system and the associated network, which benefits all jobs sharing those resources.

SCR consists of two components: a library and a set of commands. The application invokes the SCR library to read and write checkpoint files, and the library maintains the checkpoint cache. The SCR commands are typically invoked from the job batch script. They are used to prepare the cache before a job starts, automate the process of restarting a job, and copy the latest checkpoint from cache to the parallel file system upon a failure. The SCR library is described in Section 4. The SCR commands are discussed in Section 5. Important concepts applicable to both are covered in Section 3. Details on configuring an SCR job are provided in Section 6, and instructions on how to stop an SCR job are provided in Section 7. However, one must first understand all of the assumptions made in the current SCR implementation as specified in the next section.

2 Assumptions

A number of assumptions are made in the SCR implementation. If any of these assumptions do not hold for a particular application, that application cannot use SCR. If this is the case, or if you have any questions, please notify the SCR developers. The goal is to expand the implementation to support a large number of applications.

1. The code must be an MPI application.
2. The code must take globally-coordinated checkpoints written primarily as a file per process.
3. A process having a particular MPI rank is only guaranteed access to its own checkpoint files, i.e., a process of a given MPI rank may not access checkpoint files written by a process having a different MPI rank within the same run or across different runs. Note that this may limit the effectiveness of the library for codes that are capable of restarting from a checkpoint with a different number of processes than were used to write the checkpoint. Such codes can often still benefit from the scalable checkpoint capability, but not the scalable restart.
4. It must be possible to store the checkpoint files from all processes in the same directory. In particular, all files belonging to a given checkpoint must have distinct names.
5. There is no support for subdirectories within a checkpoint directory; only a single flat file space.
6. Checkpoint files cannot contain data that span multiple checkpoints. In particular, there is no support for appending data of the current checkpoint to a file containing data from a previous checkpoint. Each checkpoint file set must be self-contained.
7. On some systems, checkpoints are cached in RAM disk. This restricts usage of SCR on those machines to applications whose memory footprint leaves sufficient room to store checkpoint file data in memory simultaneously with the running application. The amount of storage needed depends on the redundancy scheme used. See Section 3.6 for details.
8. SCR maintains a set of meta data files, which it stores in a subdirectory of the directory that contains the application checkpoint files. The application must allow for these SCR meta data files to coexist with its own files.
9. To use the scalable restart capability, a job must be restarted with the same number of processes as used to write the checkpoint, and each process must only access the files it wrote during the checkpoint.
10. SCR occasionally flushes files from cache to a subdirectory on the parallel file system. SCR names and creates these subdirectories within a parent directory that is specified by the application. See Section 3.4 for details.
11. Time limits should be imposed so that the SCR library has sufficient time to flush files from cache to the parallel file system before the resource allocation expires. Additionally, care should be taken so that the run does not stop in the middle of a checkpoint. See Section 7 for details.

3 Overview

This section discusses concepts one should understand about the SCR library implementation including how it interacts with file systems.

3.1 Intro to the SCR API

This section provides an overview of how one may integrate the SCR API into an application. For a more details, see Section 4.

SCR is designed to support MPI applications that write application-level checkpoints, primarily as a file-per-process. In a given checkpoint, each process may actually write zero or more files, but the current implementation assumes that each process writes roughly the same amount of data. Before adding calls to the SCR library, existing checkpointing code for such applications may look like the following:

```
int main(int argc, char* argv[]) {
  MPI_Init(argc, argv);

  for (t = 0; t < TIMESTEPS; t++) {
    /* ... do work ... */

    /* every so often, write a checkpoint */
    if (t % CHECKPOINT_FREQUENCY == 0)
      checkpoint();
  }

  MPI_Finalize();
  return 0;
}

void checkpoint() {
  /* rank 0 creates a directory on the file system,
   * and then each process saves its state to a file */

  /* get rank of this process */
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* rank 0 creates directory on parallel file system */
  if (rank == 0)
    mkdir(checkpoint_dir);

  /* hold all processes until directory is created */
  MPI_Barrier(MPI_COMM_WORLD);

  /* build file name of checkpoint file for this rank */
  char checkpoint_file[256];
  sprintf(checkpoint_file, "%s/rank_%d".ckpt",
    checkpoint_dir, rank
  );

  /* each rank opens, writes, and closes its file */
  FILE* fs = open(checkpoint_file, "w");
  if (fs != NULL) {
    fwrite(checkpoint_data, ..., fs);
    fclose(fs);
  }
```

```
}
```

The following code exemplifies the changes necessary to integrate SCR. Each change is numbered for further discussion below.

```
int main(int argc, char* argv[]) {
  MPI_Init(argc, argv);

  /**** change #1 ****/
  SCR_Init();

  for (t = 0; t < TIMESTEPS; t++) {
    /* ... do work ... */

    /**** change #2 ****/
    int need_checkpoint;
    SCR_Need_checkpoint(&need_checkpoint);
    if (need_checkpoint)
      checkpoint();
  }

  /**** change #3 ****/
  SCR_Finalize();

  MPI_Finalize();
  return 0;
}

void checkpoint() {
  /* rank 0 creates a directory on the file system,
   * and then each process saves its state to a file */

  /**** change #4 ****/
  SCR_Start_checkpoint();

  /* get rank of this process */
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /**** change #5 ****/
  /*
      if (rank == 0)
        mkdir(checkpoint_dir);

      /* hold all processes until directory is created */
      MPI_Barrier(MPI_COMM_WORLD);
  */

  /* build file name of checkpoint file for this rank */
  char checkpoint_file[256];
  sprintf(checkpoint_file, "%s/rank_%d".ckpt",
    checkpoint_dir, rank
  );

  /**** change #6 ****/
  char scr_file[SCR_MAX_FILENAME];
  SCR_Route_file(checkpoint_file, scr_file);
```

```
  /**** change #7 ****/
  /* each rank opens, writes, and closes its file */
  FILE* fs = open(scr_file, "w");
  if (fs != NULL) {
    fwrite(checkpoint_data, ..., fs);
    fclose(fs);
  }

  /**** change #8 ****/
  SCR_Complete_checkpoint(valid);
}
```

First, as shown in change #1, one must call `SCR_Init()` to initialize the SCR library before it can be used. SCR uses MPI, so SCR must be initialized after MPI has been initialized. Similarly, as shown in change #3, it is good practice to shut down the SCR library by calling `SCR_Finalize()`. This must be done before calling `MPI_Finalize()`. As shown in change #2, the application may rely on SCR to determine when to checkpoint by calling `SCR_Need_checkpoint()`. SCR can be configured with information on failure rates and checkpoint costs for the particular host platform, so this function provides a portable method to guide an application toward an optimal checkpoint frequency.

Then, as shown in change #4, the application must inform SCR when it is starting a new checkpoint by calling `SCR_Start_checkpoint()`. Similarly, it must inform SCR when it has completed the checkpoint with a corresponding call to `SCR_Complete_checkpoint()` as shown in change #8. When calling `SCR_Complete_checkpoint()`, each process sets the `valid` flag to indicate whether it wrote all of its checkpoint files successfully. SCR manages checkpoint directories, so the `mkdir` operation is removed in change #5.

Between the call to `SCR_Start_checkpoint()` and `SCR_Complete_checkpoint()`, the application must register each of its checkpoint files by calling `SCR_Route_file()` as shown in change #6. SCR "routes" the file by replacing any leading directory on the file name with a path that points to another directory in which SCR caches data for the checkpoint. As shown in change #7, the application must use the exact string returned by `SCR_Route_file()` to open its checkpoint file.

All SCR functions are collective, except for `SCR_Route_file()`. Additionally, SCR imposes the following semantics:

1. A process of a given MPI rank may only access files previously written by itself or by processes having the same MPI rank in prior runs. We say that a rank "owns" the files it writes. A process is never guaranteed access to files written by other MPI ranks.

2. During a checkpoint, a process may only access files of the current checkpoint between calls to `SCR_Start_checkpoint()` and `SCR_Complete_checkpoint()`. Once a process calls `SCR_Complete_checkpoint()` it is no longer guaranteed access to any file that it registered as part of that checkpoint via `SCR_Route_file()`.

3. During a restart, a process may only access files from its "most recent" checkpoint, and it must access those files between calls to `SCR_Init()` and `SCR_Start_checkpoint()`. That is, a process cannot access its restart files until it calls `SCR_Init()`, and once it calls `SCR_Start_checkpoint()`, it is no longer guaranteed access to its restart files. SCR selects which checkpoint is considered to be the "most recent".

These semantics enable SCR to cache checkpoint files on devices that are not globally visible to all processes, such as node-local storage. Further, these semantics enable SCR to move, reformat, or delete checkpoint files as needed, such that it can manage this cache, which may be small.

Internally, SCR duplicates `MPI_COMM_WORLD` during `SCR_Init`. MPI messages by the SCR library do not mix with messages sent by the application.

3.2 Jobs, allocations, and runs

A large-scale simulation often must be restarted multiple times in order to run to completion. It may be interrupted due to a failure, or it may be interrupted due to time limits imposed by the resource scheduler. We use the term *allocation* to refer to an assigned set of compute resources that are available to the user for a period of time. A resource manager typically assigns an id to each resource allocation, which we refer to as the *allocation id*. SCR uses the allocation id in some directory and file names. Within an allocation, a user may execute a simulation one or more times. We call each execution a *run*. For MPI applications, each run corresponds to a single invocation of `mpirun` or its equivalent. Finally, multiple allocations may be required to complete a given simulation. We refer to this series of one or more allocations as a *job*. To summarize, one or more runs occur within an allocation, and one or more allocations occur within a job.

3.3 Group, store, and redundancy descriptors

The SCR library must group processes of the parallel job in various ways. For example, if power supply failures are common, it is necessary to identify the set of processes that share a power supply. Similarly, it is necessary to identify all processes that can access a given storage device, such as an SSD mounted on a compute node. To represent these groups, the SCR library uses a *group descriptor*. Details of group descriptors are given in Section 6.2.

Each group is given a unique name. The library creates two groups by default: `NODE` and `WORLD`. The `NODE` group consists of all processes on the same compute node, and `WORLD` consists of all processes in the run. The user or system administrator can create additional groups via configuration files (Section 6).

The SCR library must also track details about each class of storage it can access. For each available storage class, SCR needs to know the associated directory prefix, the group of processes that share a device, the capacity of the device, and other details like whether the associated file system can support directories. SCR tracks this information in a *store descriptor*. Each store descriptor refers to a group descriptor, which specifies how processes are grouped with respect to that class of storage. For a given storage class, it is assumed that all compute nodes refer to the class using the same directory prefix. Each store descriptor is referenced by its directory prefix.

The library creates one store descriptor by default: `/tmp`. The assumption is made that `/tmp` is mounted as a local file system on each compute node. On Linux clusters, `/tmp` is often RAM disk or a local hard drive. Additional store descriptors can be defined by the user or system administrator in configuration files (Section 6).

Finally, SCR defines *redundancy descriptors* to associate a redundancy scheme with a class of storage devices and a group of processes that are likely to fail at the same time. It also tracks details about the particular redundancy scheme used, and the frequency with which it should be applied. Redundancy descriptors reference both store and group descriptors.

The library creates a default redundancy descriptor. It assumes that processes on the same node are likely to fail at the same time. It also assumes that checkpoints can be cached in `/tmp`, which is assumed to be storage local to each compute node. It applies an `XOR` redundancy scheme using a group size of 8. Additional redundancy descriptors may be defined by the user or system administrator in configuration files (Section 6).

3.4 Control, cache, and prefix directories

SCR manages numerous files and directories to cache checkpoint data and to record its internal state. There are three fundamental types of directories: control, cache, and prefix directories. For a detailed illustration of how these files and directories are arranged, see the example presented in Section 3.5.

The *control directory* is where SCR writes files to store internal state about the current run. This directory is expected to be stored in node-local storage. SCR writes multiple, small files in the control directory, and it may access these files frequently. It is best to configure this directory to be stored in a node-local RAM disk.

To construct the full path of the control directory, SCR incorporates a control base directory name along with the user name and allocation id associated with the resource allocation. This enables multiple users, or multiple jobs by the same user, to run at the same time without conflicting for the same control directory. The control base directory is hard-coded into the SCR library at configure time, but this value may be overridden via a system configuration file. The user may not change the control base directory.

SCR directs the application to write checkpoint files to subdirectories within a *cache directory*. SCR also stores its redundancy data in these subdirectories. The device serving the cache directory must be large enough to hold the data for one or more checkpoints plus the associated redundancy data. Multiple cache directories may be utilized in the same run, which enables SCR to use more than one class of storage within a run (e.g., RAM disk and SSD). Cache directories should be located on scalable storage.

To construct the full path of a cache directory, SCR incorporates a cache base directory name with the user name and the allocation id associated with the resource allocation. A set of valid cache base directories is hard-coded into the SCR library at configure time, but this set can be overridden in a system configuration file. Out of this set, the user may select a subset of cache base directories to use during a run. A cache directory may be the same as the control directory.

The user must configure the maximum number of checkpoints that SCR should keep in each cache directory. It is up to the user to ensure that the capacity of the device associated with the cache directory is large enough to hold the specified number of checkpoints.

SCR refers to each application checkpoint as a *dataset*. SCR assigns a unique sequence number to each dataset called the *dataset id*. It assigns dataset ids starting from 1 and counts up with each successive dataset written by the application. Within a cache directory, a dataset is written to its own subdirectory called the *dataset directory*.

Finally, the *prefix directory* is a directory on the parallel file system that the user specifies. SCR copies datasets to the prefix directory for permanent storage (Section 3.9). The prefix directory should be accessible from all compute nodes, and the user must ensure that the prefix directory is unique for each job. For each dataset stored in the prefix directory, SCR creates and manages a *dataset directory*. The dataset directory holds all files associated with a particular dataset, including application files and SCR redundancy files. SCR maintains an index file within the prefix directory, which records information about each dataset stored there.

Note that the term "dataset directory" is overloaded. In some cases, we use this term to refer to a directory in cache and in other cases we use the term to refer to a directory within the prefix directory on the parallel file system. In any particular case, the meaning should be clear from the context.

3.5 Example of SCR files and directories

To illustrate how files and directories are arranged in SCR, consider the example shown in Figure 2. In this example, a user named "`user1`" runs a 4-task MPI job with one task per compute node. The base directory for the control directory is `/tmp`, the base directory for the cache directory is `/ssd`, and the prefix directory is `/p/lscratchb/user1/simulation123`. The control and cache directories are storage devices local to the compute node.

The full path of the control directory is `/tmp/user1/scr.1145655`. This is derived from the concatenation of the control base directory (`/tmp`), the user name (`user1`), and the allocation id (`1145655`). SCR keeps files to persist its internal state in the control directory, including filemap files as shown.

Similarly, the cache directory is `/ssd/user1/scr.1145655`, which is derived from the concatenation of the cache base directory (`/ssd`), the user name (`user1`), and the allocation id (`1145655`). Within the cache directory, SCR creates a subdirectory for each dataset. In this example, there are two datasets with ids 17 and 18. The application dataset files and SCR redundancy files are stored within their corresponding dataset directory. On the node running MPI rank 0, there is one application dataset file (`rank_0.ckpt`) and one XOR redundancy data file (`1_of_4_in_0.xor`).

Finally, the full path of the prefix directory is `/p/lscratchb/user1/simulation123`. This is a path on the parallel file system that is specified by the user. It is unique to the particular simulation the user is running (`simulation123`). The prefix directory contains dataset directories. It also contains a hidden `.scr` directory where SCR writes the index file to record info for each of the datasets (Section 8). The SCR library writes other files to this hidden directory, including the "halt" file (Section 7).

While the user provides the prefix directory, SCR defines the name of each dataset directory to be "`scr.dataset.<id>`" where `<id>` is the dataset id. In this example, there are multiple datasets stored on the parallel file system corresponding to dataset ids 10, 12, and 18. Within each dataset directory, SCR stores the files written by the application. SCR also creates a hidden `.scr` subdirectory, and this hidden directory contains redundancy files and other SCR files that are specific to the dataset.

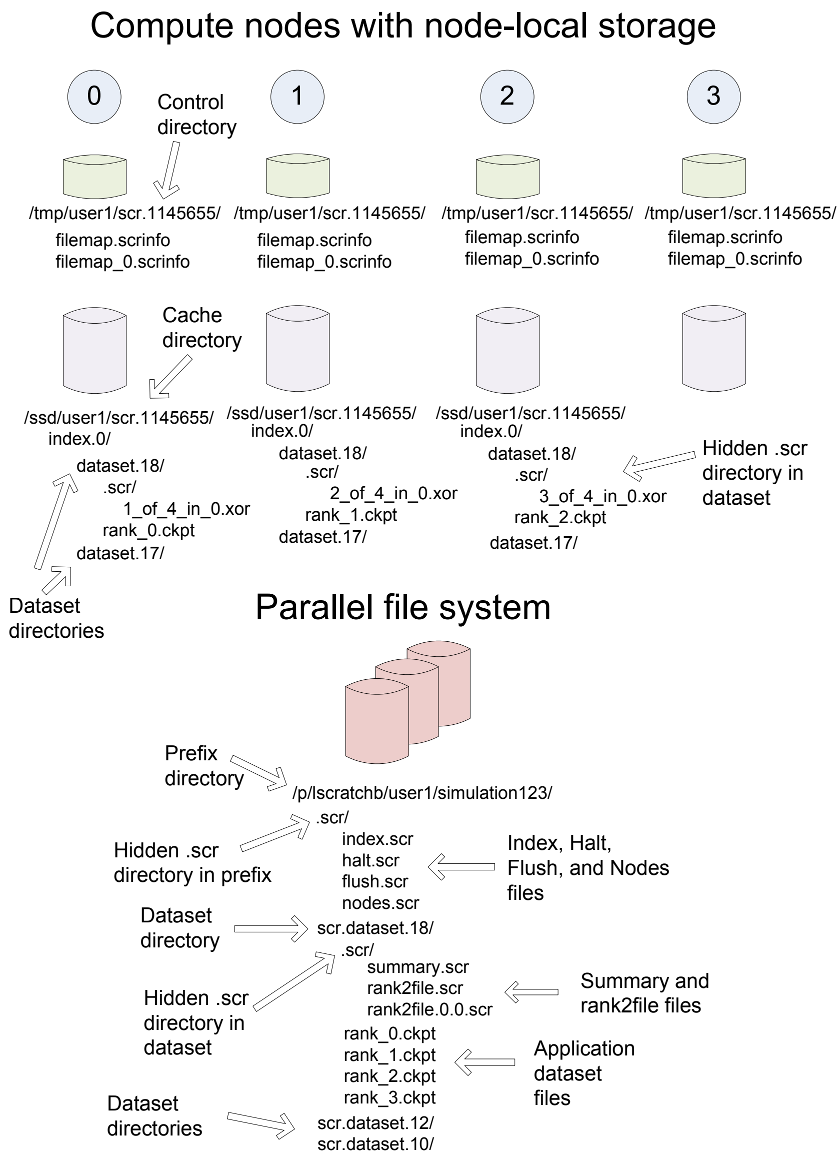

Figure 2: Example of SCR Directories

3.6 Scalable checkpoint

In practice, it is common for multiple processes to fail at the same time, but most often this happens because those processes depend on a single, failed component. It is not common for multiple, independent components to fail simultaneously. By expressing the groups of processes that are likely to fail at the same time, the SCR library can apply redundancy schemes to withstand common, multi-process failures. We refer to a set of processes likely to fail at the same time as a *failure group*.

SCR must also know which groups of processes share a given storage device. This is useful so the group can coordinate its actions when accessing the device. For instance, if a common directory must be created before each process writes a file, a single process can create the directory and then notify the others. We refer to a set of processes that share a storage device as a *storage group*.

Users and system administrators can pass information about failure and storage groups to SCR in descriptors defined in configuration files (See Section 6.2). Given this knowledge of failure and storage groups, the SCR library implements three redundancy schemes which trade off performance, storage space, and reliability:

- `Single` - each checkpoint file is written to storage accessible to the local process
- `Partner` - each checkpoint file is written to storage accessible to the local process, and a full copy of each file is written to storage accessible to a partner process from another failure group
- `XOR` - each checkpoint file is written to storage accessible to the local process, XOR parity data are computed from checkpoints of a set of processes from different failure groups, and the parity data are stored among the set.

With `Single`, SCR writes each checkpoint file in storage accessible to the local process. It requires sufficient space to store the maximum checkpoint file size. This scheme is fast, but it cannot withstand failures that disable the storage device. For instance, when using node-local storage, this scheme cannot withstand failures that disable the node, such as when a node loses power or its network connection. However, it can withstand failures that kill the application processes but leave the node intact, such as application bugs and file I/O errors.

With `Partner`, SCR writes checkpoint files to storage accessible to the local process, and it also copies each checkpoint file to storage accessible to a partner process from another failure group. This scheme is slower than `Single`, and it requires twice the storage space. However, it is capable of withstanding failures that disable a storage device. In fact, it can withstand failures of multiple devices, so long as a device and the device holding the copy do not fail simultaneously.

With `XOR`, SCR defines sets of processes where members within a set are selected from different failure groups. The processes within a set collectively compute XOR parity data which is stored in files along side the application checkpoint files. This algorithm is based on the work found in [2], which in turn was inspired by RAID5 [3]. This scheme can withstand multiple failures so long as two processes from the same set do not fail simultaneously.

Computationally, `XOR` is more expensive than `Partner`, but it requires less storage space. Whereas `Partner` must store two full checkpoint files, `XOR` stores one full checkpoint file plus one XOR parity segment, where the segment size is roughly $1/(N-1)$ times the size of a checkpoint file for a set of size N. Larger sets demand less storage, but they also increase the probability that two processes in the same set will fail simultaneously. Larger sets may also increase the cost of recovering files in the event of a failure.

3.7 Scalable restart

So long as a failure does not violate the redundancy scheme, a job can restart within the same resource allocation using the cached checkpoint files. This saves the cost of writing checkpoint files out to the parallel file system only to read them back during the restart. In addition, SCR provides support for the use of spare nodes. A job can allocate more nodes than it needs and use the extra nodes to fill in for any failed nodes during a restart. SCR includes a set of scripts which encode much of the restart logic (Section 5).

Upon encountering a failure, SCR relies on the MPI library, the resource manager, or some other external service to kill the current run. After the run is killed, and if there are sufficient healthy nodes remaining, the same job can be restarted within the same allocation. In practice, such a restart typically amounts to issuing another "`mpirun`" in the job batch script.

Of the set of nodes used by the previous run, the restarted run should use as many of the same nodes as it can to maximize the number of files available in cache. A given MPI rank in the restarted run does not need to run on the same node that it ran on in the previous run. SCR distributes cached files among processes according to the process mapping of the restarted run.

By default, SCR inspects the cache for existing checkpoints when a job starts. It attempts to rebuild all datasets in cache, and then it attempts to restart the job from the most recent checkpoint. If a checkpoint fails to rebuild, SCR deletes it from cache. To disable restarting from cache, set the `SCR_DISTRIBUTE` parameter to 0. When disabled, SCR deletes all files from cache and restarts from a checkpoint on the parallel file system.

An example restart scenario is illustrated in Figure 3 in which a 4-node job using the `Partner` scheme allocates 5 nodes and successfully restarts within the allocation after a node fails.

3.8 Catastrophic failures

There are some failures from which the SCR library cannot recover. In such cases, the application is forced to fall back to the latest checkpoint successfully written to the parallel file system. Such catastrophic failures include the following:

Multiple node failure which violates the redundancy scheme. If multiple nodes fail in a pattern which violates the cache redundancy scheme, data are irretrievably lost.

Failure during a checkpoint. Due to cache size limitations, some applications can only fit one checkpoint in cache at a time. For such cases, a failure may occur after the library has deleted the previous checkpoint but before the next checkpoint has completed. In this case, there is no valid checkpoint in cache to recover.

Failure of the node running the job batch script. The logic at the end of the allocation to scavenge the latest checkpoint from cache to the parallel file system executes as part of the job batch script. If the node executing this script fails, the scavenge logic will not execute and the allocation will terminate without copying the latest checkpoint to the parallel file system.

Parallel file system outage. If the application fails when writing output due to an outage of the parallel file system, the scavenge logic may also fail when it attempts to copy files to the parallel file system.

There are other catastrophic failure cases not listed here. Checkpoints must be written to the parallel file system with some moderate frequency so as not to lose too much work in the event of a catastrophic failure. Section 3.9 provides details on how to configure SCR to make occasional writes to the parallel file system.

By default, the current implementation stores only the most recent checkpoint in cache. One can change the number of checkpoints stored in cache by setting the `SCR_CACHE_SIZE` parameter. If space is available, it is recommended to increase this value to at least 2.

3.9 Fetch, flush, and scavenge

SCR manages the transfer of datasets between the prefix directory on the parallel file system and the cache. We use the term *fetch* to refer to the action of copying a dataset from the parallel file system to cache. When transferring data in the other direction, there are two terms used: *flush* and *scavenge*. Under normal circumstances, the library directly copies files from cache to the parallel file system, and this direct transfer is known as a flush. However, sometimes a run is killed before the library can complete this transfer. In these cases, a set of SCR commands is executed after the final run to ensure that the latest checkpoint is copied to the parallel file system before the allocation expires. We say that these scripts scavenge the latest checkpoint.

Each time an SCR job starts, SCR first inspects the cache and attempts to distribute files for a scalable restart as discussed in Section 3.7. If the cache is empty or the distribute operation fails or is disabled, SCR attempts to fetch a checkpoint from the prefix directory to fill the cache. SCR reads the index file and attempts to fetch the most recent checkpoint, or otherwise the checkpoint that is marked as current within the index file. For a given checkpoint, SCR records whether the fetch attempt succeeds or fails in the index file. SCR does not attempt to

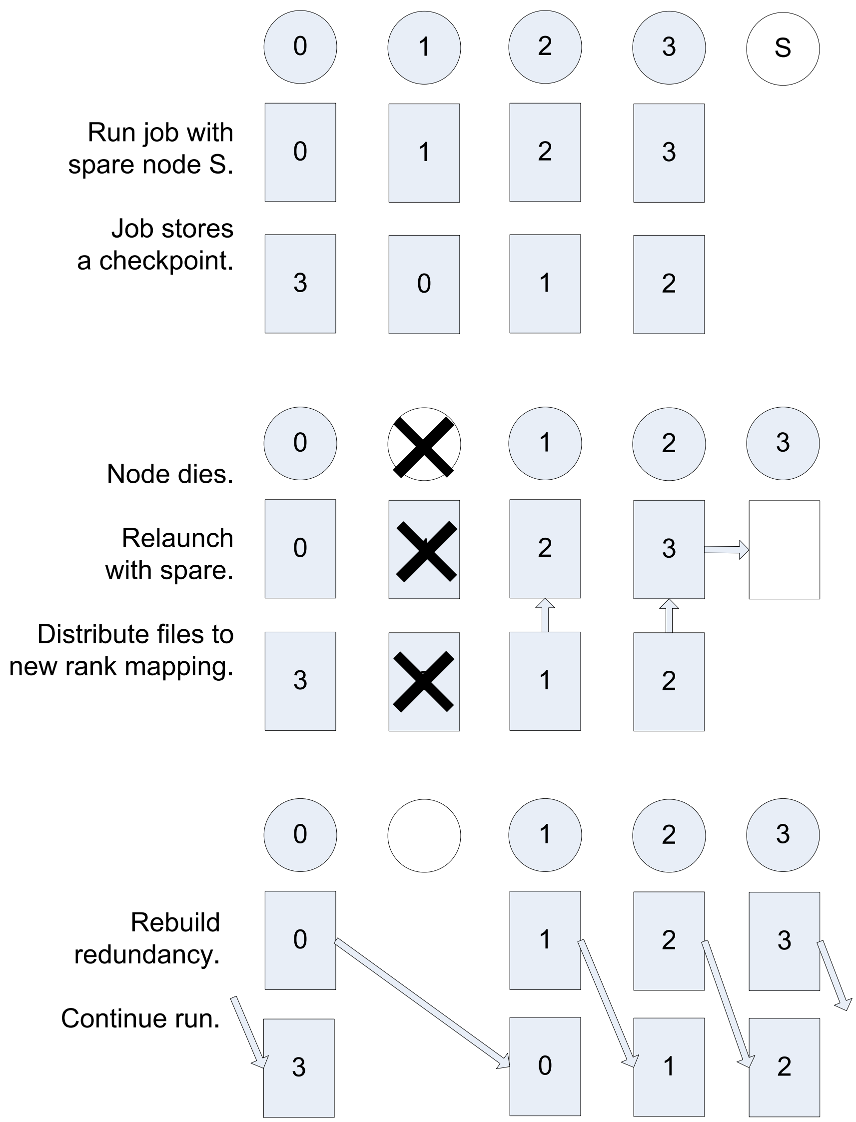

Figure 3: Example restart after a failed node with `Partner`

fetch a checkpoint that is marked as being incomplete nor does it attempt to fetch a checkpoint for which a previous fetch attempt has failed. If SCR attempts but fails to fetch a checkpoint, it prints an error and continues the run.

To disable the fetch operation, set the `SCR_FETCH` parameter to 0. If an application disables the fetch feature, the application is responsible for reading its checkpoint set directly from the parallel file system upon a restart.

To withstand catastrophic failures, it is necessary to write checkpoint sets out to the parallel file system with some moderate frequency. In the current implementation, the SCR library writes a checkpoint set out to the parallel file system after every 10 checkpoints. This frequency can be configured by setting the `SCR_FLUSH` parameter. When this parameter is set, SCR decrements a counter with each successful checkpoint. When the counter hits 0, SCR writes the current checkpoint set out to the file system and resets the counter to the value specified in `SCR_FLUSH`. SCR preserves this counter between scalable restarts, and when used in conjunction with `SCR_FETCH`, it also preserves this counter between fetch and flush operations such that it is possible to maintain periodic checkpoint writes across runs. Set `SCR_FLUSH` to 0 to disable periodic writes in SCR. If an application disables the periodic flush feature, the application is responsible for writing occasional checkpoint sets to the parallel file system.

By default, SCR computes and stores a CRC32 checksum value for each checkpoint file during a flush. It then uses the checksum to verify the integrity of each file as it is read back into cache during a fetch. If data corruption is detected, SCR falls back to fetch an earlier checkpoint set. To disable this checksum feature, set the `SCR_CRC_ON_FLUSH` parameter to 0.

4 The SCR library

This sections describes how to use the SCR library.

4.1 SCR API

The SCR API is designed to support a common checkpointing technique used by large-scale codes, which is to take globally-coordinated checkpoints written primarily as a file per process. The API is designed to be simple, scalable, and portable. It consists of a small number of function calls to wrap existing application checkpoint logic. In most cases, one may fully integrate SCR into an application with fifteen to twenty lines of source code.

Unless otherwise stated, SCR functions are collective, meaning all processes must call the function synchronously. The underlying implementation may or may not be synchronous, but to be portable, an application must treat a collective call as though it is synchronous. This constraint enables the SCR implementation to utilize the full resources of the job in a collective manner to optimize performance at critical points such as computing redundancy data.

Applications written in C should include "`scr.h`", and Fortran should include "`scrf.h`". All functions return `SCR_SUCCESS` if successful.

4.1.1 SCR_Init

```
int SCR_Init();
```

```
SCR_INIT(IERROR)
  INTEGER IERROR
```

Initialize the SCR library: identify failure groups, prepare the cache, distribute and rebuild files upon a restart, etc. This function must be called after `MPI_Init`, and it is good practice to call this function immediately after `MPI_Init`. A process should only call `SCR_Init` once during its execution. No other SCR calls are valid until a process has returned from `SCR_Init`.

In the current implementation, configuration parameters are processed during `SCR_Init`, process groups are established for redundancy computation, and internal cache and control directories are created. The library also attempts to acquire the latest checkpoint for the job. It first inspects the cache for existing checkpoints as discussed in Section 3.7. If this fails, it attempts to fetch the latest checkpoint from the parallel file system as discussed in Section 3.9. If that also fails, the library prints a warning and continues the run. Before returning from `SCR_Init`, MPI rank 0 determines whether the job should be halted and signals this condition to all other ranks (Section 7). If the job should be halted, rank 0 records a reason in the halt file, and then all tasks call `exit`, in which case, control does not return to the application.

4.1.2 SCR_Finalize

```
int SCR_Finalize();

SCR_FINALIZE(IERROR)
  INTEGER IERROR
```

Shut down the SCR library: free resources, flush datasets to the prefix directory, etc. This function must be called before `MPI_Finalize`, and it is good practice to call this function just before `MPI_Finalize`. A process should only call `SCR_Finalize` once during its execution.

If `SCR_FLUSH` is enabled, `SCR_Finalize` flushes any datasets to the prefix directory if necessary. It updates the halt file to indicate that `SCR_Finalize` has been called. This halt condition prevents the job from restarting (Section 7).

4.1.3 SCR_Need_checkpoint

```
int SCR_Need_checkpoint(int* flag);

SCR_NEED_CHECKPOINT(FLAG, IERROR)
  INTEGER FLAG, IERROR
```

Since the failure frequency and the cost of checkpointing vary across platforms, `SCR_Need_checkpoint` provides a portable way for an application to determine whether a checkpoint should be taken. This function is passed a pointer to an integer in `flag`. Upon returning from `SCR_Need_checkpoint`, `flag` is set to the value `1` if a checkpoint should be taken, and it is set to `0` otherwise. The call returns the same value in `flag` on all processes.

In addition to guiding an application to the optimum checkpoint frequency on a given platform, this call also enables a running application to react to external commands. For instance, if the application has been instructed to halt, then `SCR_Need_checkpoint` may indicate that a checkpoint should be taken.

4.1.4 SCR_Start_checkpoint

```
int SCR_Start_checkpoint();

SCR_START_CHECKPOINT(IERROR)
  INTEGER IERROR
```

Inform SCR that a new checkpoint is about to start. A process must call this function before it opens any files belonging to the new checkpoint. `SCR_Start_checkpoint` must be called by all processes, including processes that do not write files as part of the checkpoint. This function should be called as soon as possible when initiating a checkpoint. The SCR implementation uses this call as the starting point to time the cost of the checkpoint in order to optimize the checkpoint frequency via `SCR_Need_checkpoint`. Each call to `SCR_Start_checkpoint` must be followed by a corresponding call to `SCR_Complete_checkpoint`.

In the current implementation, `SCR_Start_checkpoint` holds all processes at an `MPI_Barrier` to ensure that all processes are ready to start the checkpoint before it deletes cached files from a previous checkpoint.

4.1.5 SCR_Complete_checkpoint

```
int SCR_Complete_checkpoint(int valid);

SCR_COMPLETE_CHECKPOINT(VALID, IERROR)
  INTEGER VALID, IERROR
```

Inform SCR that all files for the current checkpoint are complete (i.e., done writing and closed) and whether they are valid (i.e., written without error). A process must close all checkpoint files before calling `SCR_Complete_checkpoint`. `SCR_Complete_checkpoint` must be called by all processes, including processes that did not write any files as part of the checkpoint. The parameter `valid` should be set to `1` if either the calling process wrote all of its files successfully or it wrote no files during the checkpoint. Otherwise, the process should call `SCR_Complete_checkpoint` with `valid` set to `0`. SCR will determine whether all processes wrote their checkpoint files successfully. The SCR implementation uses this call as the stopping point to time the cost of the checkpoint that started with the preceding call to `SCR_Start_checkpoint`. Each call to `SCR_Complete_checkpoint` must be preceded by a corresponding call to `SCR_Start_checkpoint`.

In the current implementation, SCR applies the redundancy scheme during `SCR_Complete_checkpoint`. Before returning from the function, MPI rank 0 determines whether the job should be halted and signals this condition to all other ranks (Section 7). If the job should be halted, rank 0 records a reason in the halt file, and then all tasks call `exit`, in which case, control is returned to the application.

4.1.6 SCR_Route_file

```
int SCR_Route_file(const char* name, char* file);
```

```
SCR_ROUTE_FILE(NAME, FILE, IERROR)
  CHARACTER*(*) NAME, FILE
  INTEGER IERROR
```

A process calls `SCR_Route_file` to obtain the full path and file name it must use to access a checkpoint file. The name of the checkpoint file that the process intends to access must be passed in the `name` argument. A pointer to a character buffer of at least `SCR_MAX_FILENAME` bytes must be passed in `file`. When a call to `SCR_Route_file` returns `SCR_SUCCESS`, the full path and file name to access the file named in `name` is written to the buffer pointed to by `file`. The process must use the character string returned in `file` to access the file. A process does not need to create any directories listed in the string returned in `file`; the SCR implementation creates any necessary directories before returning from the call. If `SCR_SUCCESS` is not returned by a call to `SCR_Route_file`, the value in `file` is undefined and should not be used. A call to `SCR_Route_file` is local to the calling process; it is not a collective call. The precise behavior of `SCR_Route_file` depends on the calling context as described below.

During a checkpoint, a process must call `SCR_Route_file` after it calls `SCR_Start_checkpoint` and before it calls `SCR_Complete_checkpoint` to register a file as part of the current checkpoint set and to obtain the full path and file name to access the file. It is assumed the file will be opened for writing. SCR returns the same string in `file` if a process calls `SCR_Route_file` multiple times with the same `name` within a given `SCR_Start_checkpoint` / `SCR_Complete_checkpoint` pair. Once a process calls `SCR_Complete_checkpoint`, SCR may relocate, reformat, or delete checkpoint files, so paths returned by `SCR_Route_file` are no longer valid.

When using SCR during a restart, a process must call `SCR_Route_file` after `SCR_Init` and before `SCR_Start_checkpoint` to obtain the full path and file name to open a file. If a file cannot be accessed for reading, `SCR_Route_file` returns a value other than `SCR_SUCCESS`, and the string returned in `file` is undefined. When successful, `SCR_Route_file` always returns the full path and file name of the most recent checkpoint of a given file.

In the current implementation, SCR only changes the directory portion of `name`. It strips any directory components listed in `name` down to the base file name. Then, it prepends a directory to the base file name and returns the full path and file name in `file`.

4.2 Integrating the SCR API

There are three steps to consider when integrating the SCR API into an application: Init/Finalize, Checkpoint, and Restart. One may employ the scalable checkpoint capability of SCR without the scalable restart capability. While it is most valuable to utilize both, some applications cannot use the scalable restart.

4.2.1 Init/Finalize

You must add calls to SCR_Init and SCR_Finalize in order to start up and shut down the library. The SCR library uses MPI internally, and all calls to SCR must be from within a well defined MPI environment, i.e., between MPI_Init and MPI_Finalize. It is recommended to call SCR_Init immediately after MPI_Init and to call SCR_Finalize just before MPI_Finalize. For example, modify the source to look something like this:

```
// Include the SCR header
#include ‘‘scr.h’’

...

// Initialization
MPI_Init(...);
SCR_Init();

...

// Finalization
SCR_Finalize();
MPI_Finalize();
```

Some applications contain multiple calls to MPI_Finalize. In such cases, be sure to account for each call. The same applies to MPI_Init if there are multiple calls to this function.

4.2.2 Checkpoint

First, know that the application may rely on SCR to determine how often it should checkpoint. SCR can be configured with details on the failure frequency of the host platform, and it can compute the checkpoint cost, so it has sufficient information to determine the optimal checkpoint frequency. For this, the application should call SCR_Need_checkpoint at each natural opportunity it has to checkpoint, e.g., at the end of each time step, and then initiate a checkpoint when SCR advises it to do so. In the current implementation, the library makes the decision on MPI rank 0, and it broadcasts this decision to the rest of the processes. As such, this call involves some amount of global synchronization.

An application may ignore the output of SCR_Need_checkpoint, and it does not have to call the function at all. The intent of SCR_Need_checkpoint is to provide a portable way for an application to determine when to checkpoint across platforms with different reliability characteristics and different file system speeds. This function also serves as an interface to receive instructions from commands external to the job. For example, it can be used to inform the application to write a checkpoint just before the run is shut down via an external command like scr_halt (Section 7).

To actually write a checkpoint, there are three steps. First, the application must call SCR_Start_checkpoint to define the start boundary of a new checkpoint. It must do this before it opens any file it writes as part of the checkpoint. Then, the application must call SCR_Route_file for each file it writes as part of the checkpoint to register each file and to determine the full path and file name to open each file. Finally, it must call SCR_Complete_checkpoint to define the end boundary of the checkpoint.

If a process does not write any files during a checkpoint, it must still call SCR_Start_checkpoint and SCR_Complete_checkpoint as these functions are collective. All files registered through a call to SCR_Route_file between a given SCR_Start_checkpoint and SCR_Complete_checkpoint pair are considered to be part of the same checkpoint file set. Some example SCR checkpoint code looks like the following:

```
// Include the SCR header
#include ‘‘scr.h’’

...

// Determine whether we need to checkpoint
int flag;
SCR_Need_checkpoint(&flag);
if (flag) {
  // Tell SCR that a new checkpoint is starting
  SCR_Start_checkpoint();

  // Define the checkpoint file name for this process
  int rank;
  char name[256];
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  sprintf(name, ‘‘rank_%d.ckpt’’, rank);

  // Register our file, and get the full path to open it
  char file[SCR_MAX_FILENAME];
  SCR_Route_file(name, file);

  // Open, write, and close the file
  int valid = 0;
  FILE* fs = open(file, "w");
  if (fs != NULL) {
    valid = 1;
    size_t n = fwrite(checkpoint_data, 1, sizeof(checkpoint_data), fs);
    if (n != sizeof(checkpoint_data)) { valid = 0; }
    if (fclose(fs) != 0) { valid = 0; }
  }

  // Tell SCR that this process is done writing its checkpoint files
  SCR_Complete_checkpoint(valid);
}
```

In particular, note that the application should not create any directories as part of this checkpoint. The SCR library will manage directory creation. The application only must list and write its files.

4.2.3 Restart

There are two options to access files during a restart: with and without SCR. If an application is designed to restart such that each MPI task only needs access to the files it wrote during the previous checkpoint, then the application can utilize the scalable restart capability of SCR. This enables the application to restart from a cached checkpoint in the existing resource allocation, which saves the cost of writing to and reading from the parallel file system.

To use SCR, the application must call `SCR_Route_file` to determine the full path and file name to each of its checkpoint files. If this call succeeds, it returns the most recent version of the checkpoint file. It should call `SCR_Route_file` after `SCR_Init` and before `SCR_Start_checkpoint`. Some example SCR restart code may look like the following:

```
// Include the SCR header
#include ``scr.h''

...

// Initialize SCR
SCR_Init();

// Define the checkpoint file name for this process
int rank;
char name[256];
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
sprintf(name, ``rank_%d.ckpt'', rank);

// Get the full path to open our file
char file[SCR_MAX_FILENAME];
if (SCR_Route_file(name, file) == SCR_SUCCESS) {
  // Open, read, and close the file
  FILE* fs = open(file, "r");
  size_t n = fread(checkpoint_data, 1, sizeof(checkpoint_data), fs);
  fclose(fs);
} else {
  // There is no existing file to restart from
}
```

If the application does not use SCR for restart, it should not make calls to `SCR_Route_file` during the restart. Instead, it should access files directly from the parallel file system. The application must account for the SCR directory structure used to store checkpoint sets as described in Section 3.4. Be aware that this directory structure may change between SCR versions. Also, know that when restarting without SCR, the value of the `SCR_FLUSH` counter will not be preserved between restarts. The counter will be reset to its upper limit with each restart. Thus, each restart may introduce some fixed offset in a series of periodic SCR flushes. When not using SCR for restart, one should set the `SCR_FLUSH_ON_RESTART` parameter to `1`, which will cause SCR to flush any cached checkpoint to the file system during `SCR_Init`.

4.3 Building with the SCR library

To compile and link with the SCR library, add the flags in Table 1 to your compile and link lines. The value of the variable `SCR_INSTALL_DIR` should be the path to the installation directory for SCR.

Table 1: SCR build flags

Compile Flags	`-I$(SCR_INSTALL_DIR)/include`
C Dynamic Link Flags	`-L$(SCR_INSTALL_DIR)/lib -lscr -Wl,-rpath,$(SCR_INSTALL_DIR)/lib`
C Static Link Flags	`-L$(SCR_INSTALL_DIR)/lib/ -lscr -lscr_base -lscr -lz`
Fortran Dynamic Link Flags	`-L$(SCR_INSTALL_DIR)/lib -lscrf -Wl,-rpath,$(SCR_INSTALL_DIR)/lib`
Fortran Static Link Flags	`-L$(SCR_INSTALL_DIR)/lib -lscrf -lscr_base -lscrf -lz`

On LLNL systems, the default installation location (`SCR_INSTALL_DIR`) is `/usr/local/tools/scr-1.1/`. Note that some libraries are listed multiple times to satisfy cyclical link dependencies.

5 Run a job

In addition to the SCR library, one must properly configure the batch system and include a set of SCR commands in the job batch script. In particular, one must: 1) inform the batch system that the allocation should remain available even after a failure, and 2) replace the command to execute the application with an SCR wrapper script. The precise set of options and commands to use depends on the system resource manager.

The SCR commands prepare the cache, scavenge files from cache to the parallel file system, and check that the scavenged dataset is complete among other things. These commands are located in the `/bin` directory where SCR is installed. There are numerous SCR commands. Any command not mentioned in this document is not intended to be executed by users.

5.1 Supported platforms

At the time of this writing, SCR supports several machines from LLNL and Los Alamos National Laboratory (LANL). Systems similar to those described here may also work with SCR, or may work with some changes. Please contact us for help in porting SCR to other platforms. (See Section 9 for contact information)

The platforms currently supported by SCR are LLNL and LANL TLCC2 machines, which use the MOAB workload manager and the SLURM resource manager, and the LANL Cray XT platforms Cielito and Cielo, which use the MOAB workload manager and the ALPS resource manager. The descriptions for using SCR in this section apply to these specific systems, however the following description is helpful to understand how to run SCR on any system.

5.2 Jobs and job steps

First, we differentiate between a *job allocation* and a *job step*. Our terminology originates from the SLURM resource manager, but the principles apply generally across SCR-supported resource managers.

When a job is scheduled resources on a system, the batch script executes inside of a job allocation. The job allocation consists of a set of nodes, a time limit, and a job id. The job id can be obtained by executing the `squeue` command on SLURM or the `apstat` command on ALPS.

Within a job allocation, a user may run one or more job steps, each of which is invoked by a call to `srun` (SLURM) or `aprun` (ALPS). Each job step is assigned its own step id. On SLURM, within each job allocation, job step ids start at 0 and increment with each issued job step. Job step ids can be obtained by passing the `-s` option to `squeue`. A fully qualified name of a SLURM job step consists of: `jobid.stepid`. For instance, the name `1234.5` refers to step id 5 of job id 1234. On ALPS, each job step within an allocation has a unique id that can be obtained through `apstat`.

5.3 Ignoring node failures

Before running an SCR job, it is necessary to configure the job allocation to withstand node failures. By default, MOAB will kill the job allocation if a node fails, however SCR requires the job allocation to remain active in order to restart the job or to scavenge files. To enable the job allocation to continue past node failures, one must specify the appropriate flags from Table 2.

Table 2: SCR job allocation flags

MOAB batch script	`#MSUB -l resfailpolicy=ignore`
MOAB via `mxterm`	`mxterm ... -l resfailpolicy=ignore`
MOAB interactive	`qsub -I ... -l resfailpolicy=ignore`
Interactive SLURM	`salloc --no-kill ...`

5.4 The SCR wrapper script

The easiest way to integrate SCR into a batch script is to set some environment variables and to replace the job run command with an SCR wrapper script. The SCR wrapper script includes logic to restart an application within an job allocation, and it scavenges files from cache to the parallel file system at the end of an allocation. On SLURM, replace `srun` with `scr_srun`, and on ALPS, replace `aprun` with `scr_aprun`.

```
SLURM:  scr_srun [srun_options] <prog> [prog_args ...]
ALPS:  scr_aprun [aprun_options] <prog> [prog_args ...]
```

For applications that cannot invoke the SCR wrapper script as described here, one should examine the logic contained in the script and duplicate the necessary parts in the job's batch script.

The SCR wrapper script must run from within a job allocation. The command processes no parameters – it passes all parameters directly to `srun` (SLURM) or `aprun` (ALPS). Internally, the command must know the prefix directory. By default, it uses the current working directory. One may specify a different prefix directory by setting the `SCR_PREFIX` parameter.

It is recommended to set the `SCR_HALT_SECONDS` parameter so that the job allocation does not expire before the latest checkpoint can be flushed (Section 7).

By default, the SCR wrapper script does not restart an application after the first job step exits. To automatically restart a job step within the current allocation, set the `SCR_RUNS` environment variable to the maximum number of runs to attempt. For an unlimited number of attempts, set this variable to `-1`.

After a job step exits, the wrapper script checks whether it should restart the job. If so, the script sleeps for some time to give nodes in the allocation a chance to clean up. Then, it checks that there are sufficient healthy nodes remaining in the allocation. By default, the wrapper script assumes the next run requires the same number of nodes as the previous run, which is recorded in a file written by the SCR library. If this file cannot be read, the command assumes the application requires all nodes in the allocation. Alternatively, one may override these heuristics and precisely specify the number of nodes needed by setting the `SCR_MIN_NODES` environment variable to the number of required nodes.

Some applications cannot run via wrapper scripts.

5.5 Example batch script for using SCR restart capability

An example MOAB / SLURM batch script with `scr_srun` is shown below:

```
#!/bin/bash
#MSUB -l partition=atlas
#MSUB -l nodes=66
#MSUB -l resfailpolicy=ignore

# above, tell MOAB / SLURM to not kill the job allocation upon a node failure
# also note that the job requested 2 spares -- it uses 64 nodes but allocated 66

# add the scr commands to the job environment
. /usr/local/tools/dotkit/init.sh
use scr-1.1

# specify where checkpoint directories should be written
export SCR_PREFIX=/p/lscratchb/username/run1/checkpoints

# instruct SCR to flush to the file system every 20 checkpoints
export SCR_FLUSH=20

# halt if there is less than an hour remaining (3600 seconds)
export SCR_HALT_SECONDS=3600

# attempt to run the job up to 3 times
export SCR_RUNS=3

# run the job with scr_srun
scr_srun -n512 -N64 ./my_job
```

6 Configure a job

The default SCR configuration suffices for many Linux clusters. However, significant performance improvement or additional functionality may be gained via custom configuration.

6.1 Setting parameters

SCR searches the following locations in the following order for a parameter value, taking the first value it finds.

1. Environment variables,
2. User configuration file,
3. System configuration file,
4. Compile-time constants.

Some parameters, such as the location of the control directory, cannot be specified by the user. Such parameters must be either set in the system configuration file or hard-coded into SCR as compile-time constants.

To find a user configuration file, SCR looks for a file named `.scrconf` in the prefix directory (note the leading dot). Alternatively, one may specify the name and location of the user configuration file by setting the `SCR_CONF_FILE` environment variable at run time, e.g.,

```
export SCR_CONF_FILE=~/myscr.conf
```

The location of the system configuration file is hard-coded into SCR at build time. On LLNL systems, one may find this file at `/etc/scr.conf`.

To set an SCR parameter in a configuration file, list the parameter name followed by its value separated by an '=' sign. Blank lines are ignored, and any characters following the '#' comment character are ignored. For example, a configuration file may contain something like the following:

```
>>: cat ~/myscr.conf
# set the halt seconds to one hour
SCR_HALT_SECONDS=3600

# set SCR to flush every 20 checkpoints
SCR_FLUSH=20
```

6.2 Group, store, and checkpoint descriptors

SCR must have information about process groups, storage devices, and redundancy schemes. The defaults provide reasonable settings for Linux clusters, but one can define custom settings via group, store, and checkpoint descriptors in configuration files.

SCR must know which processes are likely to fail at the same time (failure groups) and which processes access a common storage device (storage groups). By default, SCR creates a group of all processes in the job called `WORLD` and another group of all processes on the same compute node called `NODE`. If more groups are needed, they can be defined in configuration files with entries like the following:

```
GROUPS=zin1  POWER=psu1  SWITCH=0
GROUPS=zin2  POWER=psu1  SWITCH=1
GROUPS=zin3  POWER=psu2  SWITCH=0
GROUPS=zin4  POWER=psu2  SWITCH=1
```

Group descriptor entries are identified by a leading `GROUPS` key. Each line corresponds to a single compute node, where the hostname is the value of the `GROUPS` key. There must be one line for every compute node in the allocation. It is recommended to specify groups in the system configuration file.

The remaining values on the line specify a set of group name / value pairs. The group name is the string to be referenced by store and checkpoint descriptors. The value can be an arbitrary character string. The only requirement is that for a given group name, nodes that form a group must specify identical values.

In the above example, there are four compute nodes: zin1, zin2, zin3, and zin4. There are two groups defined: `POWER` and `SWITCH`. Nodes zin1 and zin2 belong to the same `POWER` group, as do nodes zin3 and zin4. For the `SWITCH` group, nodes zin1 and zin3 belong to the same group, as do nodes zin2 and zin4.

In addition to groups, SCR must know about the storage devices available on a system. SCR requires that all processes be able to access the prefix directory, and it assumes that `/tmp` is storage local to each compute node. Additional storage can be described in configuration files with entries like the following:

```
STORE=/tmp           GROUP=NODE   COUNT=1
STORE=/ssd           GROUP=NODE   COUNT=3
STORE=/dev/persist   GROUP=NODE   COUNT=1  ENABLED=1  MKDIR=0
STORE=/p/lscratcha   GROUP=WORLD
```

Store descriptor entries are identified by a leading `STORE` key. Each line corresponds to a class of storage devices. The value associated with the `STORE` key is the directory prefix of the storage device. This directory prefix also serves as the name of the store descriptor. All compute nodes must be able to access their respective storage device via the specified directory prefix.

The remaining values on the line specify properties of the storage class. The `GROUP` key specifies the group of processes that share a device. Its value must specify a group name. The `COUNT` key specifies the maximum number of checkpoints that can be kept in the associated storage. The user should be careful to set this appropriately depending on the storage capacity and the application checkpoint size. The `COUNT` key is optional, and it defaults to the value of the `SCR_CACHE_SIZE` parameter if not specified. The `ENABLED` key enables (1) or disables (0) the store descriptor. This key is optional, and it defaults to 1 if not specified. The `MKDIR` key specifies whether the device supports the creation of directories (1) or not (0). This key is optional, and it defaults to 1 if not specified.

In the above example, there are four storage devices specified: `/tmp`, `/ssd`, `/dev/persist`, and `/p/lscratcha`. The storage at `/tmp`, `/ssd`, and `/dev/persist` specify the `NODE` group, which means that they are node-local storage. Processes on the same compute node access the same device. The storage at `/p/lscratcha` specifies the `WORLD` group, which means that all processes in the job can access the device. In other words, it is a globally accessible file system.

Finally, SCR must be configured with redundancy schemes. By default, SCR protects against single compute node failures using `XOR`, and it caches one checkpoint in `/tmp`. To specify something different, edit a configuration file to include checkpoint descriptors. Checkpoint descriptors look like the following:

```
# instruct SCR to use the CKPT descriptors from the config file
SCR_COPY_TYPE=FILE

# the following instructs SCR to run with three checkpoint configurations:
# - save every 8th checkpoint to /ssd using the PARTNER scheme
# - save every 4th checkpoint (not divisible by 8) to /ssd using XOR with
#   a set size of 8
# - save all other checkpoints (not divisible by 4 or 8) to /tmp using XOR with
#   a set size of 16
CKPT=0 INTERVAL=1 GROUP=NODE   STORE=/tmp TYPE=XOR     SET_SIZE=16
CKPT=1 INTERVAL=4 GROUP=NODE   STORE=/ssd TYPE=XOR     SET_SIZE=8
CKPT=2 INTERVAL=8 GROUP=SWITCH STORE=/ssd TYPE=PARTNER
```

First, one must set the `SCR_COPY_TYPE` parameter to "`FILE`". Otherwise, an implied checkpoint descriptor is constructed using various SCR parameters including `SCR_GROUP`, `SCR_CACHE_BASE`, `SCR_COPY_TYPE`, and `SCR_SET_SIZE`.

Checkpoint descriptor entries are identified by a leading `CKPT` key. The values of the `CKPT` keys must be numbered sequentially starting from 0. The `INTERVAL` key specifies how often a checkpoint is to be applied. For each checkpoint, SCR selects the descriptor having the largest interval value that evenly divides the internal SCR checkpoint iteration number. It is necessary that one descriptor has an interval of 1. This key is optional, and it defaults to 1 if not specified. The `GROUP` key lists the failure group, i.e., the name of the group of processes likely to fail. This key is optional, and it defaults to the value of the `SCR_GROUP` parameter if not specified. The `STORE` key specifies the directory in which to cache the checkpoint. This key is optional, and it defaults to the value of the `SCR_CACHE_BASE` parameter if not specified. The `TYPE` key identifies the redundancy scheme to be applied. This key is optional, and it defaults to the value of the `SCR_COPY_TYPE` parameter if not specified.

Other keys may exist depending on the selected redundancy scheme. For `XOR` schemes, the `SET_SIZE` key specifies the minimum number of processes to include in each `XOR` set.

6.3 SCR parameters

The table in this section specifies the full set of SCR configuration parameters.

Table 3: SCR parameters

Name	Default	Description
`SCR_HALT_SECONDS`	0	Set to a positive integer to instruct SCR to halt the job after completing a successful checkpoint if the remaining time in the current job allocation is less than the specified number of seconds.
`SCR_GROUP`	`NODE`	Specify name of failure group.
`SCR_COPY_TYPE`	`XOR`	Set to one of: `SINGLE`, `PARTNER`, `XOR`, or `FILE`.
`SCR_CACHE_BASE`	`/tmp`	Specify the base directory SCR should use to cache checkpoints.
`SCR_CACHE_SIZE`	1	Set to a non-negative integer to specify the maximum number of checkpoints SCR should keep in cache. SCR will delete the oldest checkpoint from cache before saving another in order to keep the total count below this limit.
`SCR_SET_SIZE`	8	Specify the minimum number of processes to include in an XOR set. Increasing this value decreases the amount of storage required to cache the checkpoint data. However, higher values have an increased likelihood of encountering a catastrophic error. Higher values may also require more time to reconstruct lost files from redundancy data.
`SCR_PREFIX`	$PWD	Specify the prefix directory on the parallel file system where checkpoints should be read from and written to.
`SCR_DISTRIBUTE`	1	Set to 0 to disable file distribution during `SCR_Init`.
`SCR_FETCH`	1	Set to 0 to disable SCR from fetching files from the parallel file system during `SCR_Init`.
`SCR_FETCH_WIDTH`	256	Specify the number of processes that may read simultaneously from the parallel file system.
`SCR_FLUSH`	10	Specify the number of checkpoints between periodic SCR flushes to the parallel file system. Set to 0 to disable periodic flushes.
`SCR_FLUSH_WIDTH`	256	Specify the number of processes that may write simultaneously to the parallel file system.
`SCR_FLUSH_ON_RESTART`	0	Set to 1 to force SCR to flush a checkpoint during restart. This is useful for codes that must restart from the parallel file system.
`SCR_RUNS`	1	Specify the maximum number of times the `scr_srun` command should attempt to run a job within an allocation. Set to -1 to specify an unlimited number of times.
`SCR_MIN_NODES`	N/A	Specify the minimum number of nodes required to run a job.
`SCR_EXCLUDE_NODES`	N/A	Specify a set of nodes, using SLURM node range syntax, which should be excluded from runs. This is useful to avoid particular nodes while waiting for them to be fixed by system administrators. Nodes in this list which are not in the current allocation are silently ignored.

Table 3 – continued from previous page

Name	Default	Description
`SCR_MPI_BUF_SIZE`	131072	Specify the number of bytes to use for internal MPI send and receive buffers when computing redundancy data or rebuilding lost files.
`SCR_FILE_BUF_SIZE`	1048576	Specify the number of bytes to use for internal buffers when copying files between the parallel file system and the cache.
`SCR_CRC_ON_COPY`	0	Set to 1 to enable CRC32 checks when copying files during the redundancy scheme.
`SCR_CRC_ON_DELETE`	0	Set to 1 to enable CRC32 checks when deleting files from cache.
`SCR_CRC_ON_FLUSH`	1	Set to 0 to disable CRC32 checks during fetch and flush operations.
`SCR_DEBUG`	0	Set to 1 or 2 for increasing verbosity levels of debug messages.
`SCR_WATCHDOG_TIMEOUT`	N/A	Set to the expected time (seconds) for checkpoint writes to in-system storage (See Section 7.5).
`SCR_WATCHDOG_TIMEOUT_PFS`	N/A	Set to the expected time (seconds) for checkpoint writes to the parallel file system (See Section 7.5).

7 Halt a job

There are several mechanisms to instruct a running SCR application to halt. It is often necessary to interact with the resource manager to halt a job.

7.1 scr_halt and the halt file

The recommended method to stop an SCR application is to use the `scr_halt` command. The command must be run from within the prefix directory, or otherwise, the prefix directory of the target job must be specified as an argument.

A number of different halt conditions can be specified. In most cases, the `scr_halt` command communicates these conditions to the running application via the `halt.scr` file, which is stored in the hidden `.scr` directory within the prefix directory. The SCR library reads the halt file when the application calls `SCR_Init` and each time the application calls `SCR_Complete_checkpoint`. If a halt condition is satisfied, all tasks in the application call `exit`.

7.2 Halt after next checkpoint

You can instruct an SCR job to halt after completing its next successful checkpoint:

```
scr_halt
```

To run `scr_halt` from outside of a prefix directory, specify the target prefix directory like so:

```
scr_halt /p/lscratcha/user1/simulation123
```

You can instruct an SCR job to halt after completing X checkpoints via the `--checkpoints` option. For example, to instruct a job to halt after 10 more checkpoints, use the following:

```
scr_halt --checkpoints 10
```

If the last of the X checkpoints is unsuccessful, the job continues until it completes a successful checkpoint. This ensures that SCR has a successful checkpoint to flush before it halts the job.

7.3 Halt before or after a specified time

It is possible to instruct an SCR job to halt *after* a specified time using the `--after` option. The job will halt on its first successful checkpoint after the specified time. For example, you can instruct a job to halt after "12:00pm today" via:

```
scr_halt --after '12:00pm today'
```

It is also possible to instruct a job to halt *before* a specified time using the `--before` option. For example, you can instruct a job to halt before "8:30am tomorrow" via:

```
scr_halt --before '8:30am tomorrow'
```

For the "halt before" condition to be effective, one must also set the `SCR_HALT_SECONDS` parameter. When `SCR_HALT_SECONDS` is set to a positive number of seconds, SCR checks how much time is left before the specified time limit. If the remaining time is less than or equal to `SCR_HALT_SECONDS`, SCR halts the job. The value of `SCR_HALT_SECONDS` does not affect the "halt after" condition.

It is highly recommended that `SCR_HALT_SECONDS` be set so that the SCR library can impose a default "halt before" condition using the end time of the job allocation. This ensures the latest checkpoint can be flushed before the allocation is lost.

It is important to set `SCR_HALT_SECONDS` to a value large enough that SCR has time to completely flush (and rebuild) files before the allocation expires. Consider that a checkpoint may be taken just *before* the remaining time is less than `SCR_HALT_SECONDS`. If a code checkpoints every X seconds and it takes Y seconds to flush files from the cache and rebuild, set `SCR_HALT_SECONDS` $= X + Y + \Delta$, where Δ is some positive value to provide additional slack time.

One may also set the halt seconds via the `--seconds` option to `scr_halt`. Using the `scr_halt` command, one can set, change, and unset the halt seconds on a running job.

NOTE: If any `scr_halt` commands are specified as part of the batch script before the first run starts, one must then use `scr_halt` to set the halt seconds for the job rather than the `SCR_HALT_SECONDS` parameter. The `scr_halt` command creates the halt file, and if a halt file exists before a job starts to run, SCR ignores any value specified in the `SCR_HALT_SECONDS` parameter.

7.4 Halt immediately

Sometimes, you need to halt an SCR job immediately, and there are two options for this. You may use the `--immediate` option:

```
scr_halt --immediate
```

This command first updates the halt file, so that the job will not be restarted once stopped. Then, it kills the current run.

If for some reason the `--immediate` option fails to work, you may manually halt the job.[2] First, issue a simple `scr_halt` so the job will not restart, and then manually kill the current run using mechanisms provided by the resource manager, e.g., `scancel` for SLURM and `apkill` for ALPS.

When using mechanisms provided by the resource manager to kill the current run, be careful to cancel the job step and not the job allocation. Canceling the job allocation destroys the cache.

For SLURM, to get the job step id, type: `squeue -s`. Then be sure to include the job id *and* step id in the `scancel` argument. Do *not* just type "`scancel 1234`" – be sure to include the job step id. For example, if the job id is 1234 and the step id is 5, then use the following commands:

```
scr_halt
scancel 1234.5
```

For ALPS, use `apstat` to get the apid of the job step to kill. Then, follow the steps as described above: execute `scr_halt` followed by the kill command `apkill <apid>`.

[2] On Cray/ALPS, `scr_halt --immediate` is not yet supported. The alternate method described in the text must be used instead.

7.5 Catch a hanging job

If an application hangs, SCR may not be given the chance to copy files from cache to the parallel file system before the allocation expires. To avoid losing significant compute time due to a hang, SCR attempts to detect if a job is hung, and if so, SCR attempts to kill the job step so that it can be restarted in the allocation.

On some systems, SCR employs the `io-watchdog` library for this purpose. For more information on this tool, see `http://code.google.com/p/io-watchdog` and on LLNL systems, see `/usr/local/tools/io-watchdog/README`.

On systems where `io-watchdog` is not available, SCR uses a generic mechanism based on the expected time between checkpoints as specified by the user. If the time between checkpoints is longer than expected, SCR assumes the job is hung. Two SCR parameters determine how many seconds should pass between I/O phases in an application, i.e. seconds between consecutive calls to `SCR_Start_checkpoint`. These are `SCR_WATCHDOG_TIMEOUT` and `SCR_WATCHDOG_TIMEOUT_PFS`. The first parameter specifies the time to wait when SCR writes checkpoints to in-system storage, e.g. SSD or RAM disk, and the second parameter specifies the time to wait when SCR writes checkpoints to the parallel file system. The reason for the two timeouts is that writing to the parallel file system generally takes much longer than writing to in-system storage, and so a longer timeout period is useful in that case.

When using this feature, be careful to check that the job does not hang near the end of its allocation time limit, since in this case, SCR may not kill the run with enough time before the allocation ends. If you suspect the job to be hanging and you deem that SCR will not kill the run in time, manually cancel the run as described above.

7.6 Combine, list, change, and unset halt conditions

It is possible to specify multiple halt conditions. To do so, simply list each condition in the same `scr_halt` command or issue several commands. For example, to instruct a job to halt after 10 checkpoints or before "8:30am tomorrow", which ever comes earlier, you could issue the following command:

```
scr_halt --checkpoints 10 --before '8:30am tomorrow'
```

The following sequence also works:

```
scr_halt --checkpoints 10
scr_halt --before '8:30am tomorrow'
```

You may list the current settings in the halt file with the `--list` option, e.g.,:

```
scr_halt --list
```

You may change a setting by issuing a new command to overwrite the current value.

Finally, you can unset some halt conditions by prepending `unset-` to the option names. See the `scr_halt` man page for a full listing of unset options. For example, to unset the "halt before" condition on a job, type the following:

```
scr_halt --unset-before
```

7.7 Remove the halt file

Sometimes, especially during testing, you may want to run in an existing allocation after halting a previous run. When SCR detects a halt file with a satisfied halt condition, it immediately exits. This is the desired effect when trying to halt a job, however this mechanism also prevents one from intentionally running in an allocation after halting a previous run. Along these lines, know that SCR registers a halt condition whenever the application calls `SCR_Finalize`.

When there is a halt file with a satisfied halt condition, a message is printed to `stdout` to indicate why SCR is halting. For example, you may see something like the following:

```
SCR: rank 0 on hype55: Job exiting: Reason: SCR_FINALIZE_CALLED.
```

To run in such a case, first remove the satisfied halt conditions. You can unset the conditions or reset them to appropriate values. Another approach is to remove the halt file via the `--remove` option. This deletes the halt file, which effectively removes all halt conditions. For example, to remove the halt file from a job, type:

```
scr_halt --remove
```

8 Manage datasets

SCR records the status of datasets stored on the parallel file system in the `index.scr` file. This file is stored in the hidden `.scr` directory within the prefix directory. The library updates the index file as an application runs. Before copying a dataset to the parallel file system, SCR records an entry for that dataset in the index file.

To restart a job, the SCR library reads the index file during `SCR_Init` to determine which checkpoints are available. The library starts with the most recent checkpoint and works backward. SCR does not fetch any checkpoint marked as "incomplete" or "failed". A checkpoint is marked as incomplete if it was determined to be invalid during the flush or scavenge. Additionally, the library marks a checkpoint as failed if it detected a problem during a previous fetch attempt (e.g., detected data corruption). In this way, the library skips invalid or problematic checkpoints.

One may list or modify the contents of the index file via the `scr_index` command. The `scr_index` command must run within the prefix directory, or otherwise, one may specify a prefix directory using the "`--prefix`" option. The default behavior of `scr_index` is to list the contents of the index file, e.g.:

```
>>: scr_index
   DSET VALID FLUSHED             DIRECTORY
*     18 YES   2014-01-14T11:26:06 scr.dataset.18
      12 YES   2014-01-14T10:28:23 scr.dataset.12
       6 YES   2014-01-14T09:27:15 scr.dataset.6
```

When listing datasets, the internal SCR dataset id is shown, followed by a field indicating whether the dataset is valid, the time it was flushed to the parallel file system, and finally the dataset directory name.

One checkpoint may also be marked as "current". When restarting a job, the SCR library starts from the current dataset and works backwards. The current dataset is denoted with a leading `*` character. One can change the current checkpoint using the `--current` option.

```
scr_index --current scr.dataset.12
```

In most cases, the SCR library or the SCR commands add all necessary entries to the index file. However, there are cases where they may fail. In particular, if the `scr_postrun` command successfully scavenges a dataset but the resource allocation ends before the command can rebuild missing files, an entry may be missing from the index file. In such cases, one may manually add the corresponding entry using the "`--add`" option.

When adding a new dataset directory to the index file, the `scr_index` command checks whether the files in a dataset directory constitute a complete and valid set. It rebuilds missing files if there are sufficient redundant data, and it writes the `summary.scr` file if needed.

```
scr_index --add scr.dataset.50
```

One may also remove entries from the index file using the "`--remove`" option. This operation does not delete the corresponding dataset files – it only deletes the entry from the `index.scr` file.

```
scr_index --remove scr.dataset.50
```

This is useful if one deletes a dataset from the parallel file system and then wishes to update the index.

9 Support and Contacts

The main repository for SCR is located at:

https://github.com/hpc/scr

From this site, you can download the source code and manuals for the current release of SCR.

For information about the project including active research efforts, please visit:

https://computation.llnl.gov/project/scr

To contact the developers of SCR for help with using or porting SCR, please visit:

https://computation.llnl.gov/project/scr/contact.php

There you will find links to join our discussion mailing list for help topics, and our announcement list for getting notifications of new SCR releases.

References

[1] N. H. Vaidya, "A case for two-level recovery schemes," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 656–666, 1998.

[2] W. Gropp, R. Ross, and N. Miller, "Providing Efficient I/O Redundancy in MPI Environments," in *Lecture Notes in Computer Science, 3241:7786, September 2004. 11th European PVM/MPI Users Group Meeting*, 2004.

[3] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," in *Proc. of 1988 ACM SIGMOD Conf. on Management of Data*, 1988.

[4] M. Jette, "Simple Linux Utility for Resource Management (SLURM)." https://computing.llnl.gov/linux/slurm.